Memory Management Simulator

2024년도 1학기 운영체제 프로젝트

제출일	2024. 06. 12	**전공**	전자정보공학부 전자공학전공
과목	운영체제	**학번**	20192490, 20192513 20201639, 20192468
담당교수	박재근 교수님	**이름**	김현우, 윤종민, 남현준, 권주현

목차

1. 서론

1.1 프로젝트 개요

운영체제 강의를 수강하며, 초반부에 프로세스와 스케줄링 관련한 강의와 학습은 오랜 시간 진행하였지만 메모리 관리에 관해서는 과제 수행 및 추가 학습을 깊게 하지 못하여 본 프로젝트를 진행하였다. 본 프로젝트는 메모리 관리 기능을 GUI(Graphic User Interface)를 통해 시각적으로 제공함으로써, 사용자들이 보다 쉽게 시스템의 메모리 상태를 이해하고 관리할 수 있도록 하는 것을 목표로 한다.

1.2 프로젝트 구상

한 프로세스가 실행되기 위해 도착하면 그 프로세스의 크기가 페이지 몇 개 분에 해당하는가를 조사한다. 각 사용자 페이지는 한 프레임씩이 필요하다. 즉, 프로세스가 n개 페이지를 요구하면 메모리에서 이용할 수 있는 프레임이 n개 있어야 한다. 그 다음 프로세스의 처음 페이지가 할당된 프레임 중 하나에 적재되고, 그 프레임 번호가 페이지 테이블에 기록된다. 그 다음 페이지는 또 다른 프레임에 적재되고, 또 그 프레임 번호가 페이지 테이블에 기록되며 이 과정이 반복된다.

페이징의 가장 중요한 특징은 메모리에 대한 프로그래머의 인식과 실제 내용이 서로 다르다는 것이다. 프로그래머는 메모리가 하나의 연속적인 공간이며, 메모리에는 이 프로그램만 있다고 생각한다. 그러나 실제로는 프로그램은 여러 곳에 프레임 단위로 분산되어 있고, 많은 다른 프로그램이 올라와 있다. 프로그래머가 생각하는 메모리와 실제 물리 메모리의 차이는 주소 변환 하드웨어에 의해 해소된다. 논리 주소는 물리 주소로 변환된다. 이 과정은 프로그래머에게는 안 보이고 운영체제에 의해 조정된다. 따라서 사용자 프로세스는 자기의 것이 아닌 메모리는 접근조차 할 수가 없다. 페이지 테이블을 통하지 않고 다른 공간에 접근할 길이 없으며 페이지 테이블은 그 프로세스가 소유하고 있는 페이지들만 가리키고 있기 때문이다.

프로젝트 목표

1. 메모리 사용 현황 시각화 : 현재 메모리 사용량, 할당된 메모리 블록, 캐시 상태 등을 시각적으로 보여준다.
2. 메모리 적재 대기 : 메모리 초과로 인한 적재 실패시에 대기 큐에 등록
3. 사용자 인터랙션 도구 : 메모리 할당 및 해제 등을 직접 제어할 수 있는 인터페이스를 제공한다.
4. GUI 분리에 따른 메모리 프로세스 과정 확인 : 그래픽 화면 기준 왼쪽에는 터미널, 중앙에는 메모리, 오른쪽에는 task를 분리하여 사용자가 터미널에서 커맨드 및 프로그램을 실행했을 때 메모리의 변화를 살필 수 있다.

2. 본론

2-1 Memory Management Simulator 구현

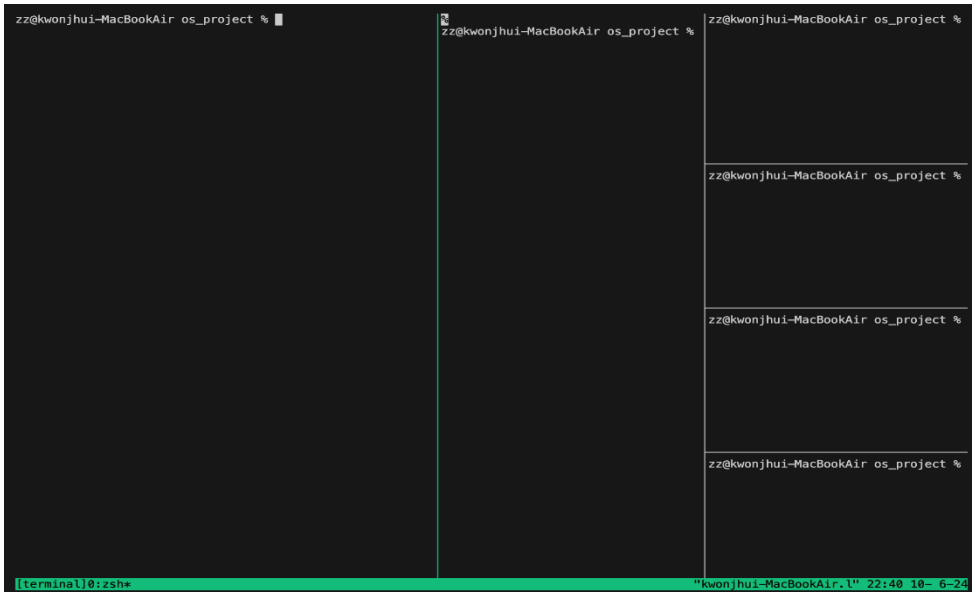
2-1.1 Boot 프로그램 및 miniOS 프로그램

부트 프로그램

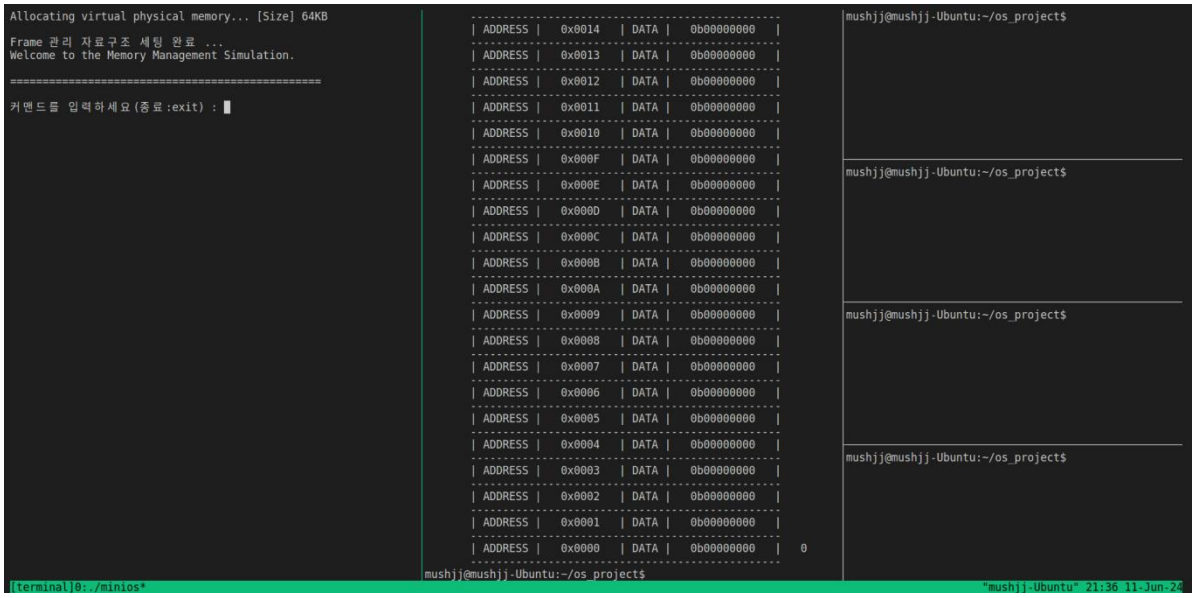
tmux를 사용하여 세션을 생성하고 창을 여러 개로 분할한 후(pane 분할), 생성한 세션에 연결하는 프로그램이다. 각 창은 수직 및 수평으로 분할되며 이는 tmux 세션 내에서 다양한 작업을 동시에 수행할 수 있는 환경을 제공한다. 소스코드의 boot.c 파일에서 자세한 구현 코드를 확인할 수 있다.

초기 터미널에서 ./powerOn 이라는 커맨드로 시뮬레이터 환경을 실행하면 여러 서비스와 커맨드 처리를 담당하는 터미널(pane 0), 메모리를 시각적으로 보며 터미널에서 작업하는 과정을 직접적으로 볼 수 있게 하는 메모리 VIEW (pane 1), 프로그램이 실행되는 Program Monitor (pane 2~5)가 차례로 tmux (terminal multiplexer)를 이용하여 분할된 화면으로 세팅 되며 miniOS 프로그램의 실행을 대기한다.

(터미널에서 ./minios 커맨드로 kernel을 로드하여 Memory Management Simulator를 실행)



[./powerOn 커맨드 입력시]



[터미널에서 ./minios 커맨드 입력시]

miniOS 프로그램

miniOS 프로그램은 커널의 역할을 한다. 본 Memory management simulator 에서 사용 가능한 모든 명령어들이 정의되어 있고 메모리 프레임과 프로그램을 관리하기 위한 수많은 자료구조들이 이를 관리하는 함수들과 함께 세팅 되어 있다.

miniOS를 실행하면 Frame list, frame manager, page manager, process pool, waiting queue를 생성하여 miniOS를 종료할 때까지 유지된다. miniOS 프로그램에서 사용가능한 명령어는 help. show_m, show_f, show_efl, show_pp, execute, terminate, exit가 있다.

help - 명령어 사용 방법을 보여 준다.

show_m - 몇 번지부터 몇 번지 메모리를 보고 싶은 지 물어보고 그 범위의 메모리를 메모리 VIEW에 업데이트 한다.

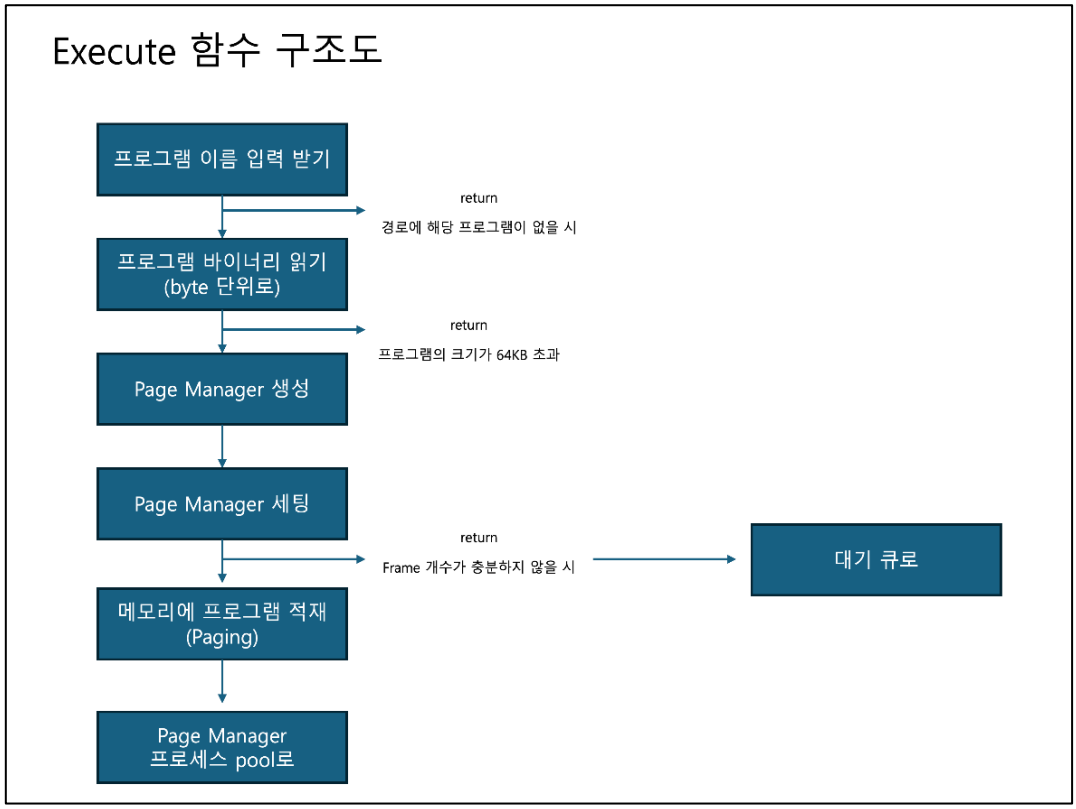
show_f - 현재 프레임 상태를 볼 수 있다.

show_efl - 현재 가용 가능 프레임 리스트를 볼 수 있다.

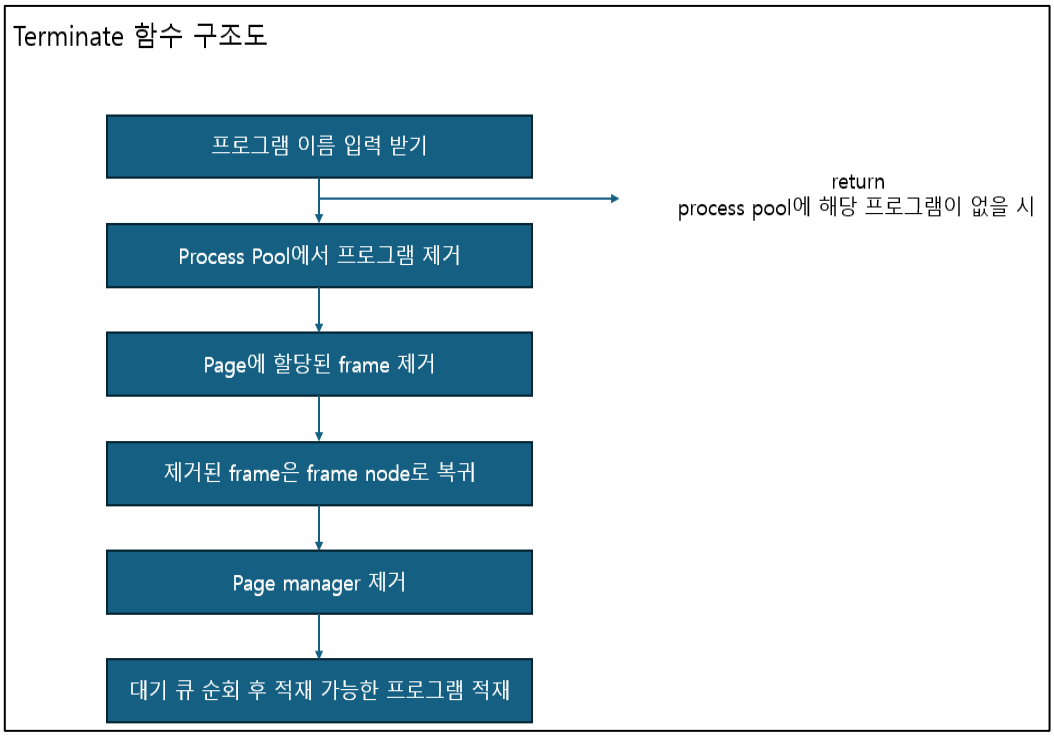
show_pp - 현재 실행 중인 프로그램의 프로세스풀을 볼 수 있다.

execute - 어떤 프로그램을 실행할지 물어보고 그 프로그램을 메모리에 적재한 후 Program Monitor 화면에서 이를 실행한다.

terminate - 어떤 프로그램을 종료할지 물어보고 그 프로그램을 Program Monitor 화면에서 종료시킨 후 메모리를 반환한다.



[execute flow chart]

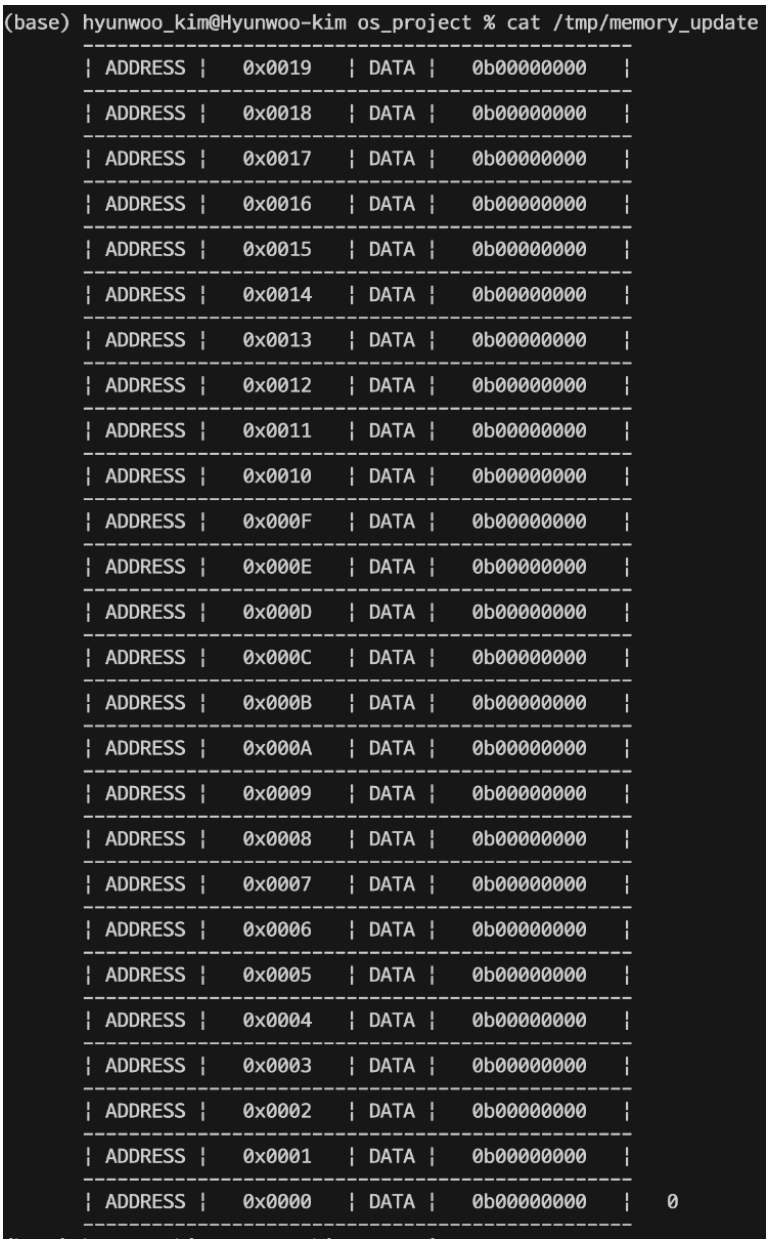


[terminate flow chart]

[execute 실행시에 메모리에 적재되는 과정]

2-1.2 Dummy Memory 제작

실제 물리메모리를 이용하기 위해 kernel 프로세스(miniOS 프로그램)에서 64KB의 메모리 공간을 동적 할당 받아 이용하는 것으로 하였다. kernel/mem_s 폴더에 있는 파일들이 이 Dummy memory를 관리하게 되는데, mem_allocate.c 파일에 있는 함수들은 메모리를 할당 받고 내부 값들을 초기화 하는 작업을 수행한다. 또한 dummy_memory_management.c 파일에 있는 함수들은 메모리 VIEW에 메모리 그림을 나타내기 위한 작업을 수행한다.



[메모리 VIEW에서 볼 수 있는 메모리 그림]

메모리 VIEW에서는 유저의 편의를 위해 실제 메모리 주소가 0x0000부터 0xFFFF의 범위로 변환되어 나타나게 하였고 각 주소마다 적재된 값을 2진수로 확인할 수 있게 하였다.

2-1.3 Memory Frame 제작

페이징(paging) 기법의 구현을 위해 메모리를 프레임을 단위로 나누었는데, 물리 메모리를 일정한 크기(4KB)의 블록으로 나누었고 이를 관리하기 위한 자료구조와 함수들을 제작하였다. 이 프레임들은 Frame Manager에 의해 유지, 관리, 보수되며 가용 가능한 프레임 리스트를 구성한다.

Physical memory frame 관련 자료구조

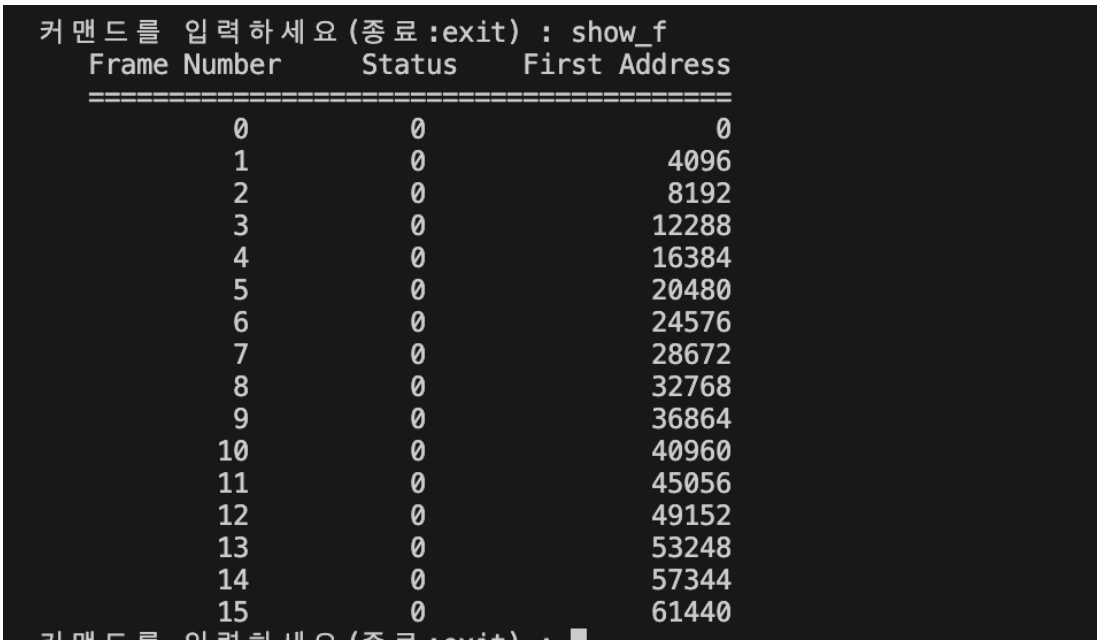
- Frame - 각 프레임 하나를 관리한다.
- Frame Manager - 모든 프레임 (16개)를 관리한다.
- Frame List - 가용 가능한 프레임 리스트를 관리한다.
- Frame Node - 가용 가능한 프레임 리스트 내의 구성 노드이다.

show_efl 명령어를 통해 (available) frame list를 확인할 수 있다.

```
커맨드를 입력하세요(종료:exit) : show_efl

   [ HEAD ]

   Frame Number:   9    First address :  36864
   Frame Number:  10    First address :  40960
   Frame Number:  11    First address :  45056
   Frame Number:  12    First address :  49152
   Frame Number:  13    First address :  53248
   Frame Number:  14    First address :  57344
   Frame Number:  15    First address :  61440

   [ TAIL ]
```

[show_efl 명령어 실행 결과]

show_f 명령어를 통해 Frame Manager가 관리하는 frame 들의 번호, 상태(페이지가 할당 되었는지), 첫번째 메모리 공간의 주소를 확인할 수 있다.

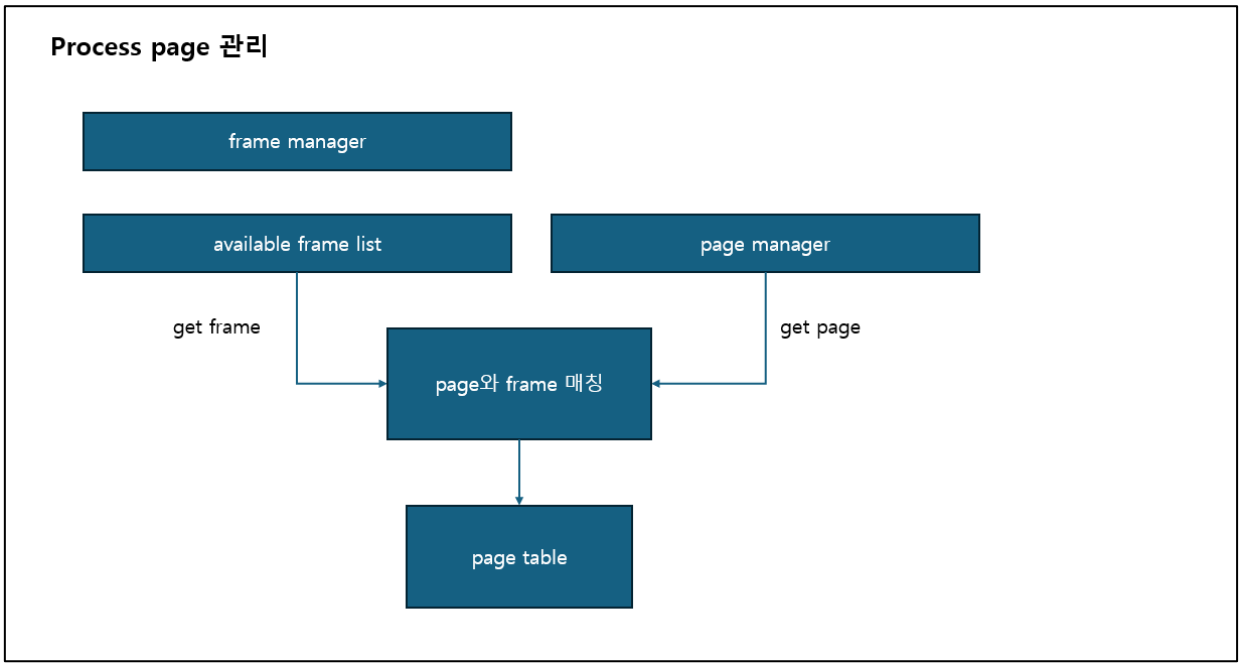


[show_f 명령어 실행 결과]

2-1.4 Process Page 관리

Paging

각 page를 frame에 매핑하는 기법이다. 이로 인해 메모리에서 생기는 단편화 문제를 줄이고 메모리 관리를 더욱 효율적으로 할 수 있다는 장점이 있다. page는 page manager를 통해 생성, 관리된다. Frame manager와 다르게 page manager는 process당 하나씩 생성된다. page manager를 생성하고 읽어온 data를 page에 채우면 page가 완성된다. 이후 available frame list에서 가져온 빈 frame을 page와 매칭한다.



[page와 frame의 matching 과정]

2-1.5 Process Load

Process Pool

성공적으로 모든 page와 frame이 매칭되어 실행하고자 하는 프로그램의 page manager가 세팅이 완료되면 프로그램을 메모리에 적재하고 실행한다. 이때 실행된 프로그램은 실행 중인 프로그램을 모아두는 프로세스 풀(Process Pool)로 들어간다. Process pool로 들어간 프로그램은 실행이 모두 완료되고 user가 terminate하기 전까지 프로세스 풀에 머무르며, terminate 될 때 프로세스 풀에서 삭제된 후 종료된다.

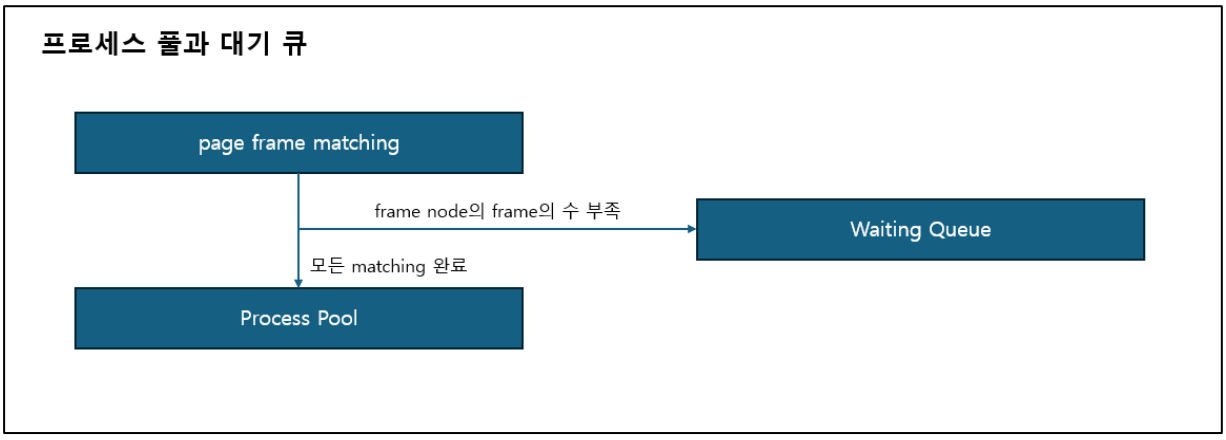
[현재 실행중인 process]

대기 큐(waiting queue)

만약 available frame list에 남아있는 빈 frame의 수가 부족하거나 메모리가 가득 차서 모든 page에 frame을 매칭시키지 못하면 해당 프로그램은 waiting queue로 간다. Waiting queue로 들어간 process는 메모리에 적재 되지 않으며 실행중인 process가 terminate 되어 가용 가능한 프레임 수가 충분해지는 등, Process Pool 에 들어가지 못한 이유가 해소되면 그때 waiting Queue를 나와 메모리에 적재되고 Process Pool로 들어가게 된다.

```
처음 20 Bytes :

CF FA ED FE 0C 00 00 01 00 00 00 00 02 00 00 00 11 00 00 00

마지막 20 Bytes :

E2 3D 3D 26 F2 74 DC 09 92 18 79 78 B2 00 00 00 00 00 00 00


[ 총 33600 byte, 9 page로 분할 ] All pages setting 완료
[ up_down_game ] Program Page manager setting 완료

Not enough Frames
 [ up_down_game ] program 이 대기 큐에 추가되었습니다

커맨드를 입력하세요(종료:exit) :
```

[메모리가 가득 찼을 때 대기 큐에 enqueue]



[Process가 상황에 따라 Process Pool 혹은 waiting queue로 가는 과정]

```
커맨드를 입력하세요(종료:exit) : terminate
종료할 프로그램을 입력하세요 : howru

[ howru ] is removed from process pool.

    [ 0번 frame 반환 ]
    [ 1번 frame 반환 ]
    [ 2번 frame 반환 ]
    [ 3번 frame 반환 ]
    [ 4번 frame 반환 ]
    [ 5번 frame 반환 ]
    [ 6번 frame 반환 ]
    [ 7번 frame 반환 ]
    [ 8번 frame 반환 ]

[ howru ] is terminated !

현재 Waiting Queue에 대기중인 프로세스 [ 1 ]개

첫번째 대기 프로세스 [ up_down_game ] dequeued!
필요한 frame 수 [ 9 ] / 현재 가용 가능한 frame 수 [ 16 ]

메모리에 적재 가능 !

 [ 0번 page => 0번 frame에 매핑 ]
 [ 1번 page => 1번 frame에 매핑 ]
 [ 2번 page => 2번 frame에 매핑 ]
 [ 3번 page => 3번 frame에 매핑 ]
 [ 4번 page => 4번 frame에 매핑 ]
 [ 5번 page => 5번 frame에 매핑 ]
 [ 6번 page => 6번 frame에 매핑 ]
 [ 7번 page => 7번 frame에 매핑 ]
 [ 8번 page => 8번 frame에 매핑 ]

| Page # | Frame # |
|---|---|
| 0 | 0 |
| 1 | 1 |
| 2 | 2 |
| 3 | 3 |
| 4 | 4 |
| 5 | 5 |
| 6 | 6 |
| 7 | 7 |
| 8 | 8 |

프로그램 적재(loading) 완료
```

[실행중인 프로그램이 terminate된 후에 대기 큐에서 대기 중이던 프로그램이 실행되는 모습]

2-2 자료구조

2-2.1 Frame block을 관리하는 자료구조

- Frame 한 개의 구조체

```
typedef struct {
    int frame_number;
    int is_allocated;
    size_t first_address;
} Frame;
```

- 각 frame 을 관리하는 manager

```
typedef struct {
    Frame frames[TOTAL_FRAMES];
    unsigned char * mem;
} FrameManager;
```

- status 가 0 인 frame 을 담는 구조체

```
typedef struct FrameNode {
    Frame frame;
    struct FrameNode * prev;
    struct FrameNode * next;
} FrameNode;
```

- FrameNode 간의 양방향 연결 리스트

```
typedef struct {
    FrameNode * head;
    FrameNode * tail;
} FrameList;
```

2-2.2 Page를 관리하는 자료구조

- Page 한 개의 구조체

```
typedef struct {
    unsigned int data[PAGE_SIZE];
    int page_number;
    int matched_frame; // 각 page 와 frame matching 된 값
    int is_matched_frame; // 각 page 와 frame matching 여부 판단
} Page;
```

- 한 process 의 page 들을 관리하는 page manager

```
typedef struct {
    Page * pages; // dynamic allocation
    int allocated_pages; // 총 몇개의 페이지인지
    int is_memory_loaded; // is making page table finished?
} PageManager;
```

2-2.3 Process Pool에 관련된 자료구조

- Process 정보를 담는 구조체

```
typedef struct Process {
    PageManager * page_manager;
    struct Process * next;
    char process_name[30];
    int pane_num;
} Process;
```

- Process Pool 관리를 위한 구조체

```
typedef struct ProcessPool {
    Process* head;
} ProcessPool;
```

2-2.4 Waiting queue에 관련된 자료구조

- waiting queue 에 들어온 program 을 담아두는 node

```
typedef struct WaitingNode {
    PageManager *page_manager;
    char process_name[20];
    struct WaitingNode *next;
} WaitingNode;
```

- waiting queue 를 연결하는 linked list 형태의 구조체

```
typedef struct WaitingQueue {
    WaitingNode *front;
    WaitingNode *rear;
} WaitingQueue;
```

2-3 Overall Flow

2-3.1 Initial setting

부트 프로그램

'./powerOn' 명령어를 통해 세션을 생성하고 창을 여러 개로 분할한 후(pane 분할), 생성한 세션에 연결한다.

miniOS 실행

'./minios' 명령어를 통해 miniOS 프로그램(Memory management simulator)을 실행하고 Terminal, 메모리 VIEW와 Program Monitor를 준비한다.

2-3.2 Simulator flow

minios 프로그램 상에서 64KB 메모리 공간을 할당 받아 이를 Paging 기법 구현을 위한 (Dummy) Physical Memory로 사용을 하도록 구현했다. Page와 Frame의 크기는 4KB 단위다.

minios 실행시 Frame Manager, (Available) Frame List, Process Pool, Waiting Queue를 생성하여 minios 종료시까지 본 프로그램에서 다루어지는 모든 자료구조, 구조체, 함수, 데이터를 유지 · 관리 · 보수한다. execute 커맨드를 통해 실행하고자 하는 프로그램을 읽어 Page 단위로 나누고 Page Manager를 세팅 (Page table을 만드는 과정이라고 보아도 된다)하여 메모리에 적재하면 Program Monitor에서 이 프로그램이 실행되는 모습을 확인할 수 있다.

프로그램이 실행 되지 않는 경우 (Program Monitor에 올라가지 않을 때)

- Program Monitor에 공간(pane)이 부족한 경우
- 할당 가능한 frame의 수가 부족할 때. 이 경우 대기큐로 들어감

프로세스 풀을 통해 현재 메모리에 적재된 프로세스들을 확인할 수 있다.

terminate 커맨드를 통해 현재 실행중인 프로그램을 종료시킬 수 있다. 프로세스 풀에서 종료하고자 하는 프로그램을 제거 하고 할당 되었던 메모리 영역을 반환한 뒤 메모리 공간의 값을 0x00으로 초기화한다. Program Monitor에서도 제거된다. (pane 공간을 반환) 모든 작업 수행 후 Waiting Queue를 한바퀴 순회하며 할당 가능한 프로세스들 (Page Manager가 모두 세팅된 상태로 Queue에 전달되기 때문에 바로 할당하는 로직만 수행해 주면 된다)을 모두 메모리에 적재한다. 이 때도 적재 불가능한 프로그램은 다시 Waiting Queue에 enqueue한다.

3. 결론

3.1 결론

본 프로젝트에서는 리눅스 커널의 메모리 관리 기능을 GUI를 통해 시각화하는 Memory Management Simulator를 개발하였다. 이 시뮬레이터는 사용자가 시스템의 메모리 사용 현황을 보다 쉽게 이해하고 관리할 수 있도록 돕는다. 가상 메모리와 물리 메모리 사이의 page, frame 관계와 page table이 제작되는 과정을 직관적으로 이해할 수 있게 해주며, 시스템 관리자들은 직관적인 대시보드를 통해 효율적으로 시스템을 모니터링할 수 있다. 또한, 운영체제의 메모리 관리 섹터를 공부할 때 부족한 부분들을 GUI로 직접 실행하고 체험함으로써 학습에 큰 도움이 된다.

이 프로젝트를 통해 사용자는 메모리 관리의 이론적인 지식을 실제로 적용해볼 수 있으며, 메모리의 동적 할당, 프레임 및 페이지 관리, 대기 큐의 처리 과정 등 운영체제의 핵심 개념을 이해할 수 있었다. 특히, 여러 프로세스가 실행되면서 메모리가 어떻게 배분되고 관리되는지를 시각적으로 확인할 수 있어, 복잡한 메모리 관리의 원리를 보다 쉽게 이해할 수 있었다.

이 프로젝트는 사용자가 메모리 관리의 복잡한 개념을 보다 명확히 이해하고, 실제 운영체제 환경에서 메모리 관리가 어떻게 이루어지는지 체험할 수 있는 유용한 도구가 될 것이다. 궁극적으로, 이러한 시뮬레이터는 교육적 도구뿐만 아니라 시스템 관리자들에게도 실질적인 도움이 될 수 있다.

3.2 Trouble shooting 및 개선 필요성

프레임, 페이지, 프로세스를 구성하는 모든 요소들이 프로그램(kernel 프로그램의 역할을 하는 minios 프로그램)이 실행되면서 계속 유지, 관리되어야 하고 데이터가 손상되면 안 되기 때문에 운영체제의 관점에서 이러한 데이터들을 관리해야 할 자료구조를 제작하는 데 초점을 두었다. 하지만 서로 엮이는 요소들이 너무나도 많아 이를 구현하기가 쉬운 일은 아니었기에 자료구조들과 이를 관리하는 알고리즘 및 함수들의 아키텍처를 설계하는 일에 시간을 많이 투자하여 시뮬레이터를 제작하였다.

본 프로젝트에서는 paging 기법에서의 페이지와 프레임의 할당, 반환 등의 기술에 초점을 맞추었다. '로더'와 '가상 메모리' 개념은 기술적인 문제로 구현하지 못하였기에 다음과 같은 요소들의 개선 필요성이 존재한다.

1. Virtual memory에서의 page swap기능 구현 : 메모리의 page를 backing storage로 swap하는 과정을 추가해 더 많은 degree of multiprogramming 을 추구하여 CPU utilization을 증가시킨다.

2. 로더 기능 추가 : 로더 기능을 추가한다. 실제 executable file이 메모리에 적재되는 과정은 로더의 도움을 받게 되고 이를 통해 메모리 관리의 효율성을 높이며 프로그램의 시작 시간을 단축할 수 있다.

추가로, 본 시뮬레이터는 그냥 명령어를 입력해서 프로세스를 메모리에 적재하고 실행하는 일련의 과정이 하나씩 순차대로 진행되지만, 실제 시스템에서는 동시 다발적으로 프로그램의 적재와 종료가 발생할 것이기에 여러 자료구조 접근에 동기화 기법을 추가하는 것도 고려해보아야 할 것이다.

* 본 보고서에서 프로그램(program)과 프로세스(process)를 혼용하여 사용하였습니다.

* 본 프로젝트의 자세한 소스 코드는 https://github.com/K-Hwoo/os_project에서 확인 가능합니다.

* 팀원별 역할은 다음과 같습니다.

김현우	팀장, Boot program 및 Kernel program 제작, Frame 관리 시스템 설계
윤종민	Page 및 page manager 관련 시스템 설계, process pool 설계
남현준	Page 및 page manager 관련 시스템 설계, 메모리 적재 알고리즘 설계
권주현	프로젝트 테스트 및 검증, 보고서 초안 작성